



Documentation

Probleme 1

Voici mes fonctions :

```
"""
    Multiplie chaque élément de la liste l par le coefficient a.

    Args:
    l (list): La liste d'entiers d'entrée.
    Returns:
    list: Une nouvelle liste contenant les éléments de l multipliés par a.

    Linear function
"""

def fctLinear(l):
    a = 2
    resultat = [element * a for element in l]
    return resultat

"""
    Calcule la somme de toutes les paires possibles d'éléments de la liste l.

    Args:
    l (list): La liste d'entiers d'entrée.
```

```

    Returns:
    int: La somme des paires d'éléments.
    """

def fctQuadratic(l):
    somme = 0
    n = len(l)
    for i in range(n):
        for j in range(n):
            somme += l[i] + l[j]
    return somme

    """
    Calcule la somme de tous les produits possibles de triplets d'éléments de la liste l.

    Args:
    l (list): La liste d'entiers d'entrée.

    Returns:
    int: La somme des produits de triplets d'éléments.
    """

def fctCubic(l):
    somme = 0
    n = len(l)
    for i in range(n):
        for j in range(n):
            for k in range(n):
                somme += l[i] * l[j] * l[k]
    return somme

    """
    Trie une liste d'entiers en utilisant l'algorithme de tri fusion (Merge Sort).

    Args:
    l (list): La liste d'entiers à trier.

    Returns:
    list: La liste triée dans l'ordre croissant.
    """

def fctPseudoLinear(l):
    if len(l) > 1:
        middle = len(l) // 2
        left_half = l[:middle]
        right_half = l[middle:]

        fctPseudoLinear(left_half)
        fctPseudoLinear(right_half)

```

```

i, j, k = 0, 0, 0

while i < len(left_half) and j < len(right_half):
    if left_half[i] < right_half[j]:
        l[k] = left_half[i]
        i += 1
    else:
        l[k] = right_half[j]
        j += 1
    k += 1

while i < len(left_half):
    l[k] = left_half[i]
    i += 1
    k += 1

while j < len(right_half):
    l[k] = right_half[j]
    j += 1
    k += 1
return l

"""
Génère toutes les permutations possibles de la liste l de manière récursive.




Args:
l (list): La liste d'éléments à permuter.



Returns:
list of list: Liste de toutes les permutations possibles.
"""

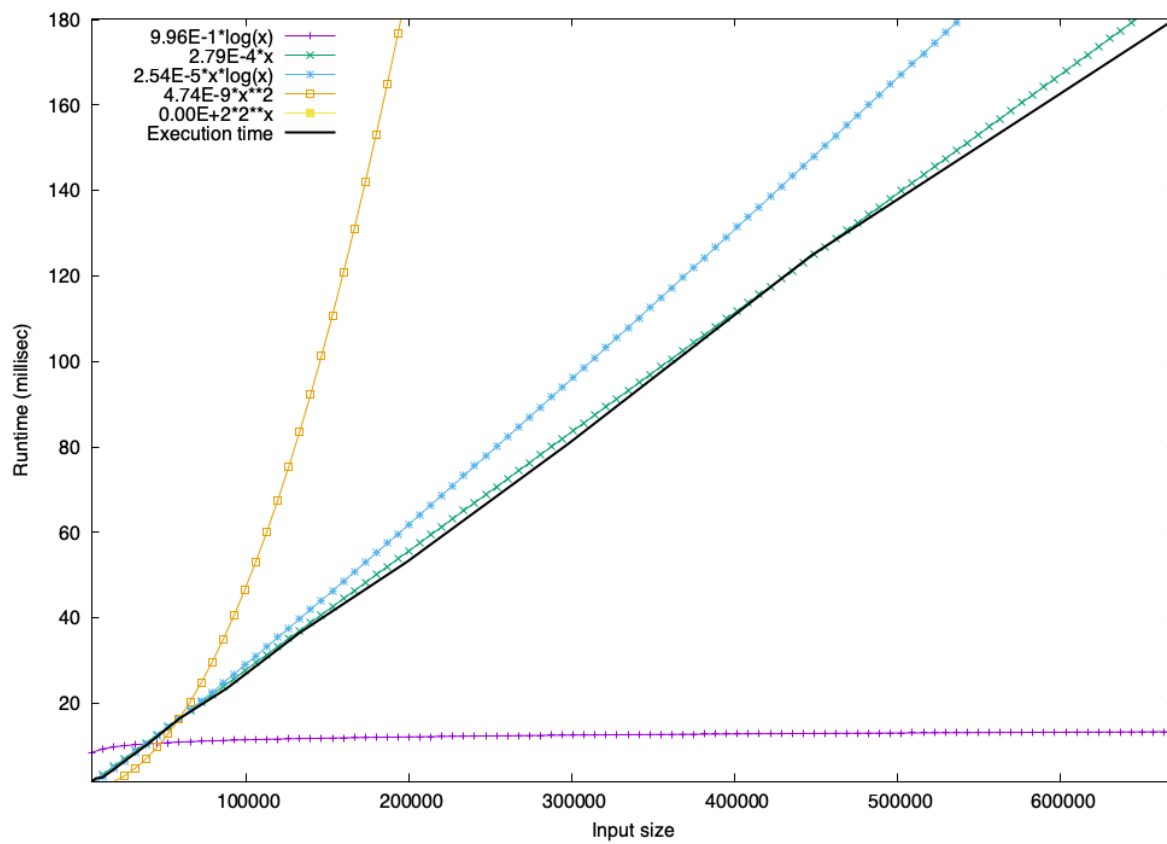
def fctExponentiel(l):
    if len(l) == 0:
        return [[]]
    subsets_without_first = fctExponentiel(l[1:])
    subsets_with_first = [[l[0]] +
                           subset for subset in subsets_without_first]
    return subsets_without_first + subsets_with_first

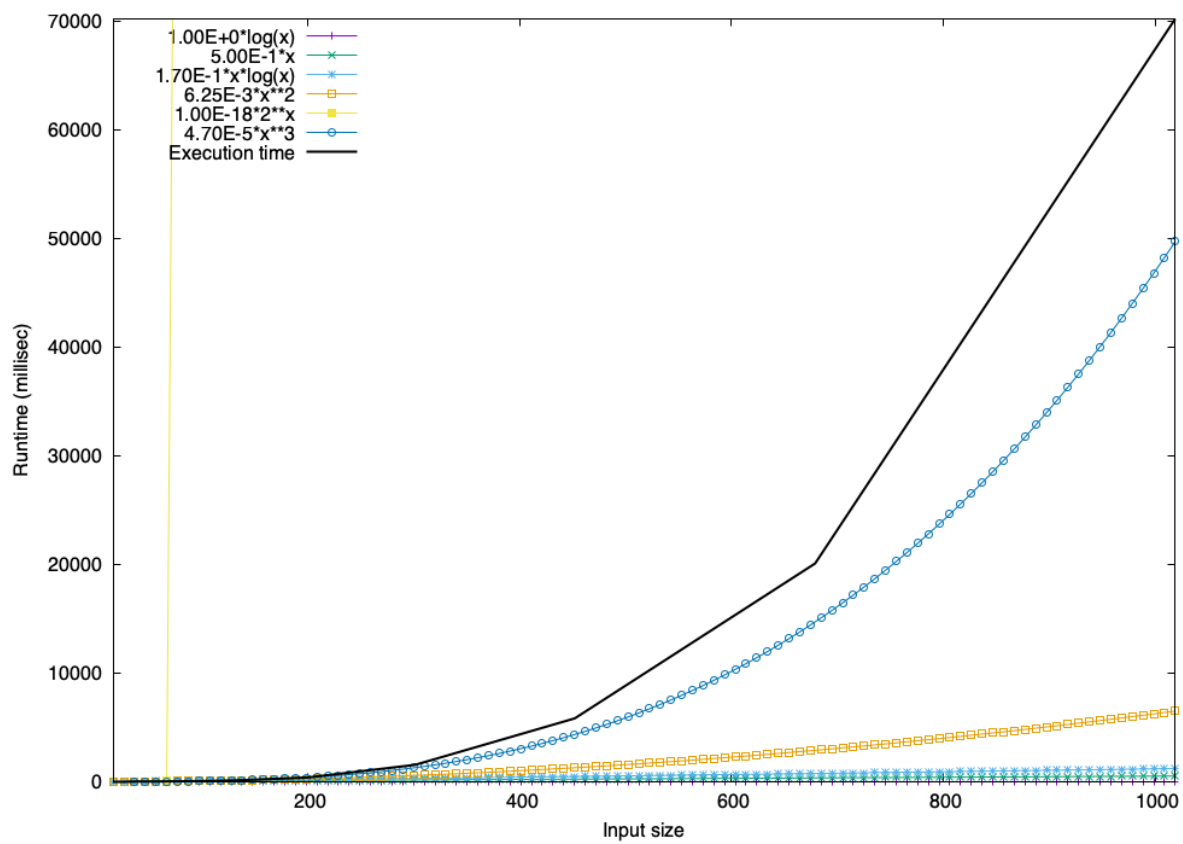
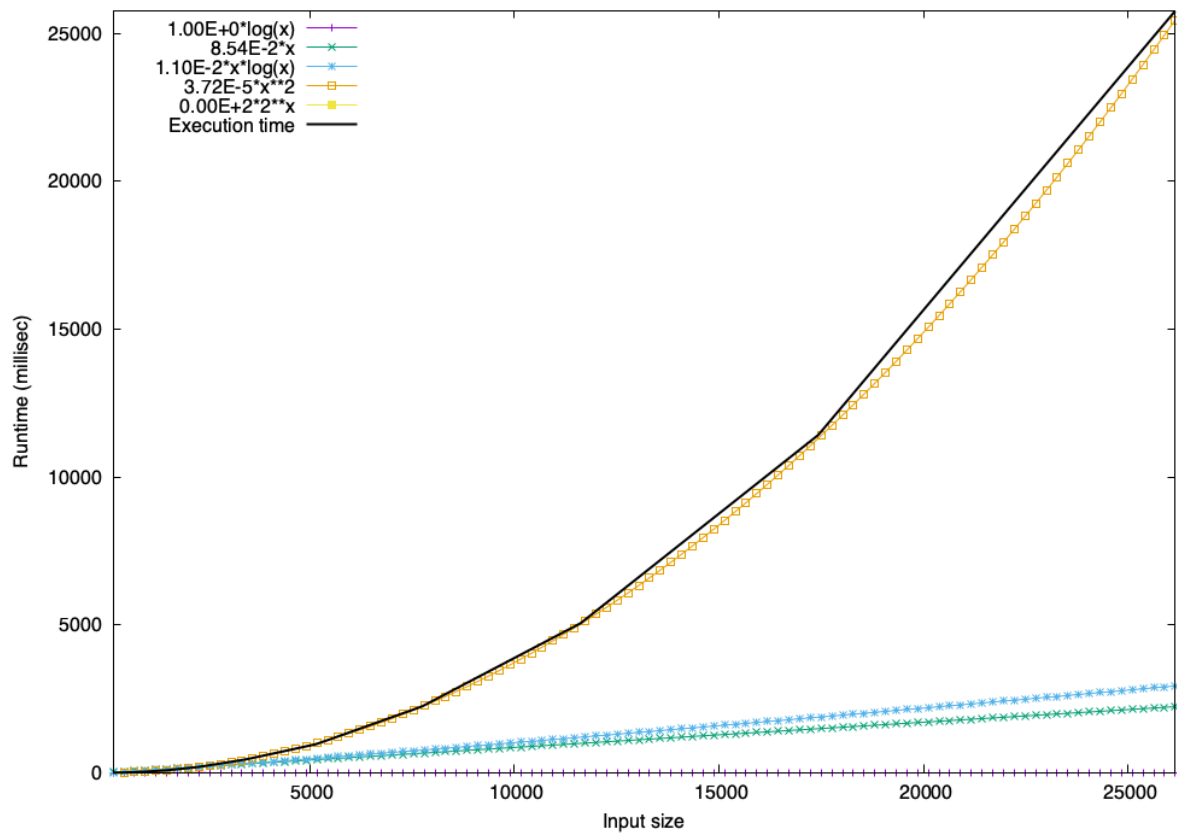
```

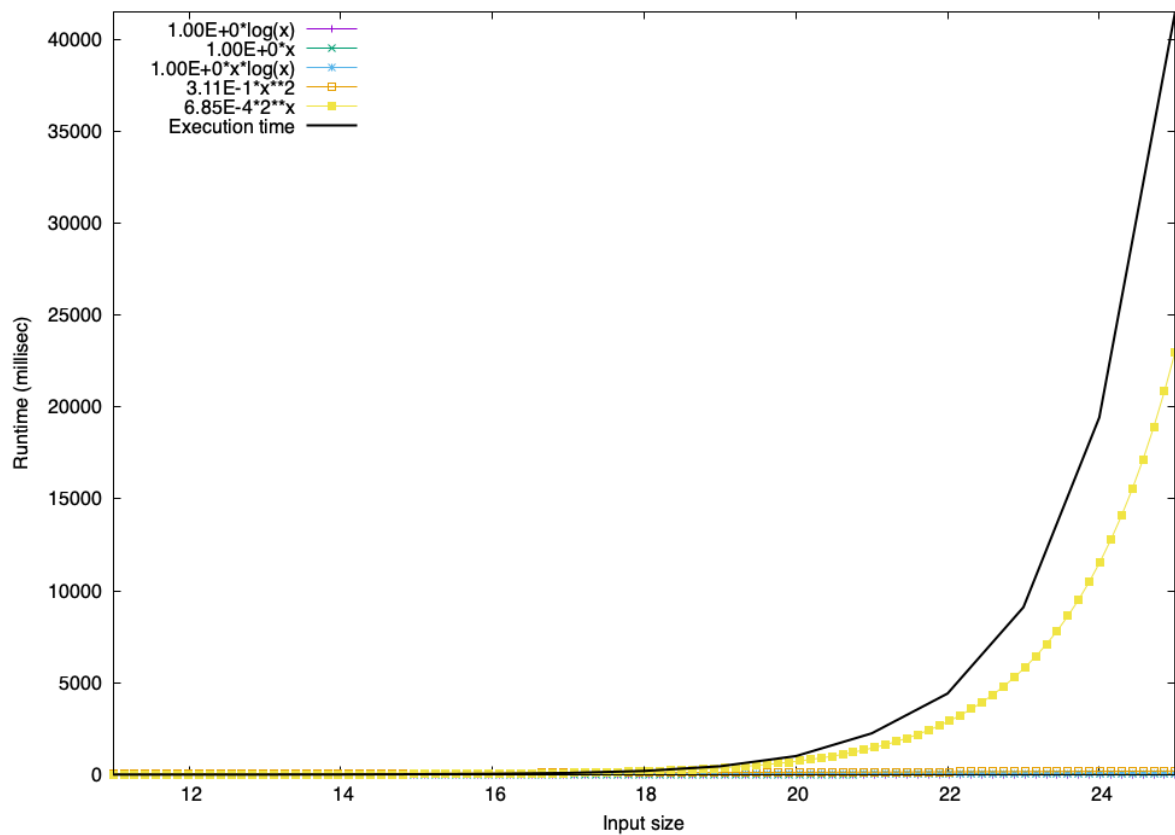
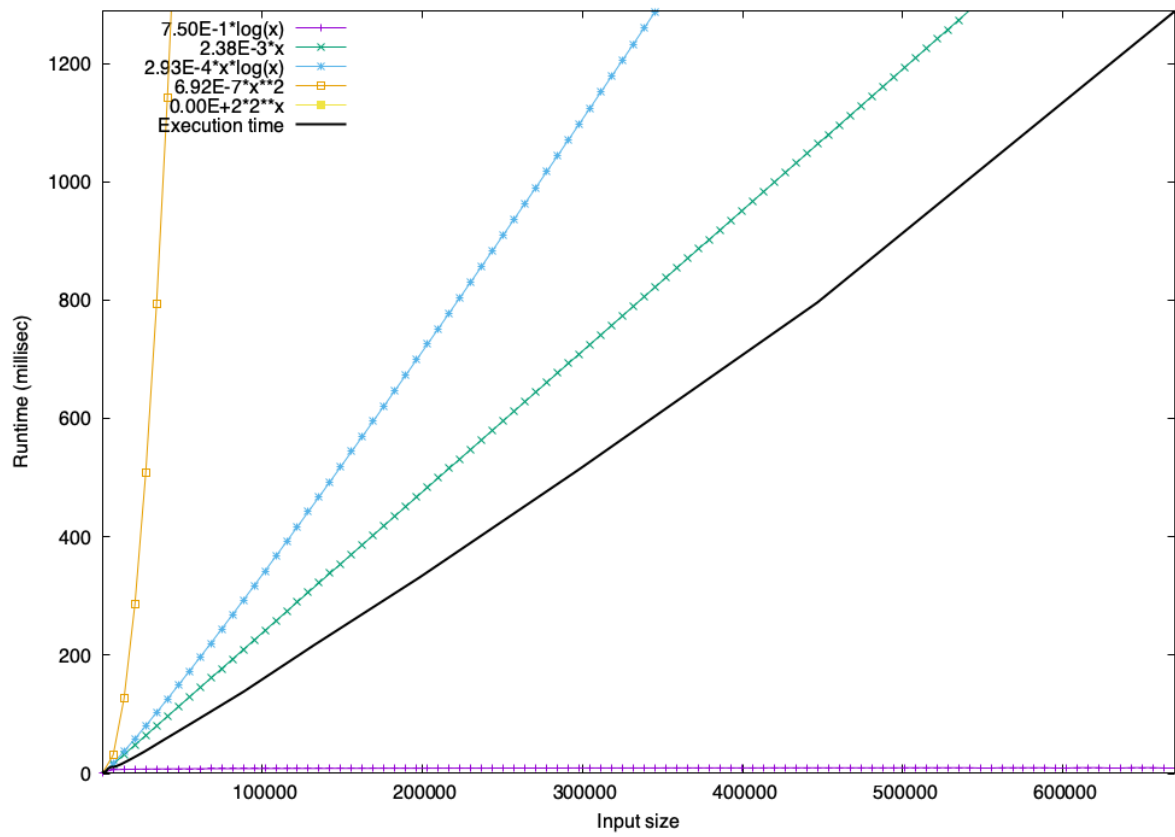
Voici les graphes :

- Linear 
- Quadratic 
- Cubic 

- PseudoLinear 
- Exponentiel 







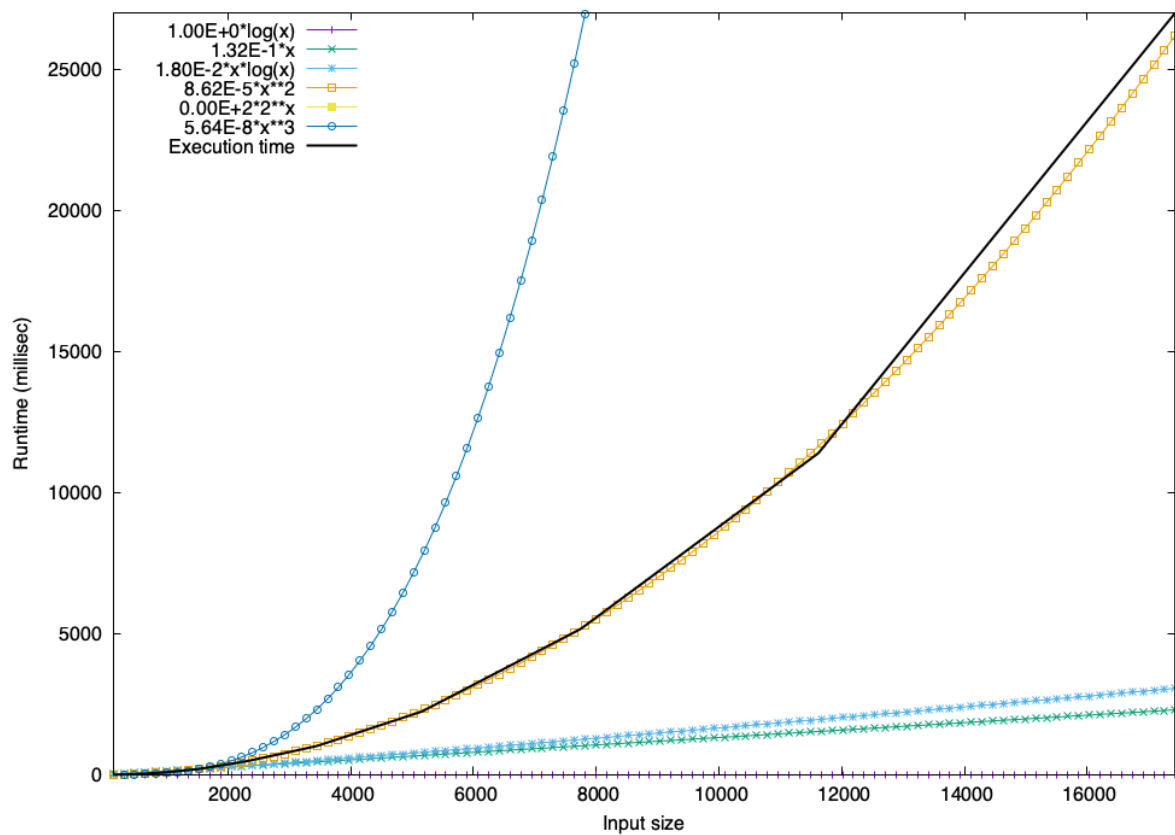
Probleme 2

a) Voici ma fonction distance min naive :

```
"""
    Du coup on retrouve une complexité en  $O(n^2)$ 
    quadratic car on parcourt deux fois la liste et on peut voir
    ceci sur le graphique dans le dossier graphs fichier algoNaiveGraph.png
"""

def distance_min_naive(L):
    if len(L) < 2:
        print("Il doit y avoir au moins deux points.")
        return 0
    else:
        daux = L[0].distance(L[1])
        for indA, a in enumerate(L):
            for indB, b in enumerate(L):
                if indA < indB:
                    if a.distance(b) < daux:
                        daux = a.distance(b)
        return daux
```

b) Voici le graphe de la fonction distance_min_naive avec une complexité quadratique :



c)

```

"""
    Trouve la distance minimale entre deux points
    dans la liste 'L' en utilisant l'algorithme de
    la distance minimale.

    Args:
    L (list of Point): Une liste de points.

    Returns:
    float: La distance minimale entre deux points.

    Reponse au question:
    - Tester la vitesse d'exécution de votre algorithme sur
    des listes de points de plus en plus rapides.
    - la complexité est  $x * \log(x) * \log(x)$  pour cette fonction
"""

def distance_min_dpr(L):
    n = len(L)

    # Si moins de 2 points, la distance minimale est infinie

```



```

if n < 2:
    return float('inf')

# Si 2 points, calculer leur distance
if n == 2:
    return L[0].distance(L[1])

# Trier les points selon leurs coordonnées x
L.sort(key=lambda point: point.x)

# Division du tableau en deux parties
mid = n // 2
L_left = L[:mid]
L_right = L[mid:]

# Récursion pour les deux moitiés
d_left = distance_min_dpr(L_left)
d_right = distance_min_dpr(L_right)

# Trouver la distance minimale entre les deux moitiés
d_min = min(d_left, d_right)

# Trouver les points proches de la ligne de séparation
strip = [point for point in L if abs(point.x - L[mid].x) < d_min]

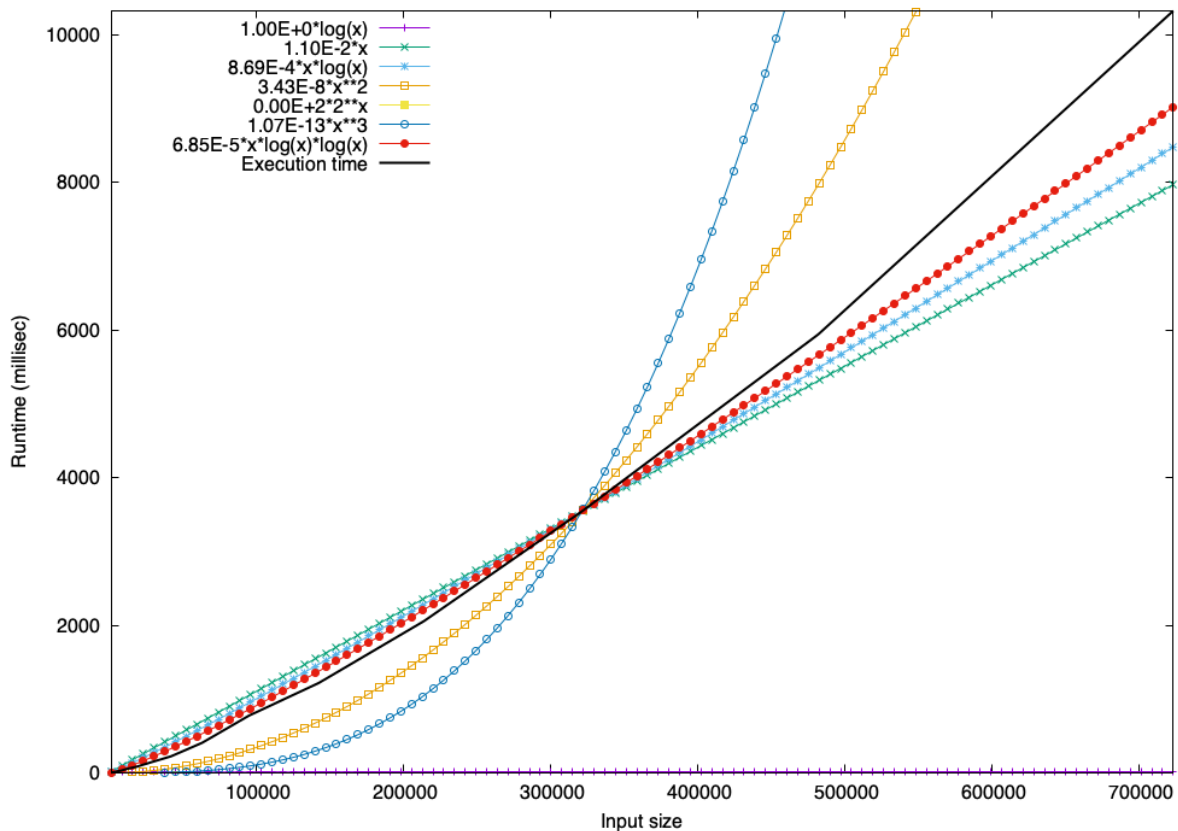
# Trier les points proches par coordonnée y
strip.sort(key=lambda point: point.y)

# Trouver la distance minimale dans la bande
for i in range(len(strip)):
    for j in range(i + 1, len(strip)):
        if strip[j].y - strip[i].y < d_min:
            dist = strip[i].distance(strip[j])
            d_min = min(d_min, dist)

return d_min

```

d)



e)

first algo : $(10^8)^2 = 1e+16$ (Nombre d'étapes de calcul)

second algo : $(10^8) * \log(10^8) * \log(10^8) = 6400000000$ (Nombre d'étapes de calcul)

mon processeur fait : 2500000000 operations par seconde

on a $1e+16 / 2500000000 = 4000000$ pour le premier algo mais on a des pertes donc

on fait $* 100 = 400000000$ secondes

on a $6400000000 / 2500000000 = 2.56$ secondes pour le second algo mais on a des

perdes donc on fait $* 100 = 256$ secondes

Probleme 3

a) Voici l'algorithme en python :

```
def sommeMin(t, n):
    return sommeMinRec(t, n)

def sommeMinRec(t, i):
```

```

    if i == 0:
        return 0

    opt = inf
    for x in [1, 3, 5]:
        if x <= i:
            tmp = t[i] + sommeMinRec(t, i - x)
            if tmp < opt:
                opt = tmp

    return opt

```

b) La taille de l'entrée est déterminée par le nombre de lignes dans le fichier. Dans votre exemple de fichier, il y a 21 lignes, dont une ligne d'en-tête et 20 lignes de valeurs. Par conséquent, la taille de l'entrée est de 20, car il y a 20 valeurs de cases dans le plateau de jeu.

La complexité temporelle de la fonction sommeMin est exponentielle en fonction de la taille de l'entrée. Cela est dû à la récursivité de la fonction sommeMinRec. Pour chaque case i , la fonction explore trois options (pas de 1, 3 ou 5) et effectue des appels récursifs pour chaque option. Cela conduit à une croissance exponentielle du nombre d'appels récursifs en fonction de la taille de l'entrée.

En d'autres termes, la complexité temporelle de la fonction sommeMin n'est pas polynomiale, mais exponentielle en fonction de la taille de l'entrée. Pour de grandes valeurs de n , le temps d'exécution de cette fonction augmentera considérablement, ce qui en fait une méthode peu efficace pour les entrées de grande taille.

c) Le temps d'exécution de l'algorithme non optimisé est de 0.16776394844055176 secondes pour résoudre le problème de taille 31. Cela confirme que le temps d'exécution est en accord avec la complexité temporelle attendue (exponentielle en $O(3^n)$) de l'algorithme non optimisé.

Justification :

L'algorithme non optimisé utilise une approche récursive où il explore toutes les combinaisons possibles de mouvements pour atteindre la case i . Pour chaque case, il explore trois options de pas (1, 3, 5), créant ainsi une récursion exponentielle. L'exponentielle est caractérisée par une croissance rapide du temps d'exécution à mesure que la taille de l'entrée augmente. Dans ce cas, pour une entrée de taille 31, le temps d'exécution est significativement plus long que pour une entrée plus petite, ce qui est conforme à une complexité exponentielle.

En résumé, le temps d'exécution de l'algorithme non optimisé augmente rapidement avec la taille de l'entrée, ce qui est caractéristique des problèmes NP-difficiles, et il correspond à la complexité exponentielle attendue.

Voici un exemple d'exécution :

Somme minimale pour atteindre la case 31 avec l'algo pas optimiser : 140
Temps d'exécution (Algo pas optimiser): 0.16776394844055176 secondes

d) L'algorithme que vous avez implémenté pour résoudre ce problème utilise une approche récursive, et il est exponentiel en termes de complexité temporelle. Voici une explication intuitive de pourquoi l'algorithme est si long :

Exponentiellement de nombreuses combinaisons : L'algorithme explore toutes les combinaisons possibles de mouvements (1, 3, ou 5 cases) pour atteindre la dernière case. Chaque fois qu'il explore une option de mouvement, il crée une branche récursive. Par conséquent, le nombre total de combinaisons à explorer augmente de manière exponentielle avec la taille de la séquence. Plus la séquence est longue, plus le nombre de branches de l'arbre de récursion augmente de manière exponentielle.

Répétition des calculs : En raison de la nature récursive de l'algorithme, de nombreux calculs sont répétés. Par exemple, lorsque vous calculez la somme minimale pour atteindre une certaine case, l'algorithme peut recalculer la somme minimale pour des cas déjà explorés à plusieurs reprises. Cela entraîne un gaspillage de ressources de calcul.

Pas d'optimisation de la mémoire : L'algorithme ne stocke pas les résultats intermédiaires, ce qui signifie qu'il peut recalculer la même chose plusieurs fois. Cela conduit à une utilisation inefficace de la mémoire et à une augmentation du temps d'exécution.

Complexité combinatoire : Le problème de minimiser la somme des valeurs en se déplaçant sur un plateau de jeu avec des contraintes de pas (1, 3, ou 5) est intrinsèquement complexe en raison des nombreuses combinaisons possibles.

En résumé, l'algorithme est long en raison de la nature exponentielle du problème, de l'exploration de nombreuses combinaisons, des calculs répétés et de l'absence d'optimisation de la mémoire. Cela signifie que le temps d'exécution augmente rapidement à mesure que la taille de la séquence augmente, ce qui est caractéristique des problèmes NP-difficiles. Pour résoudre ce problème de manière plus efficace, il serait nécessaire d'explorer des approches algorithmiques plus avancées, telles que la programmation dynamique, qui permettraient de réduire le temps d'exécution.

e)

```
def sommeMinDynamique(t, n):
    dp = [float('inf')] * (n + 1)
    dp[0] = 0

    for i in range(1, n + 1):
        for x in [1, 3, 5]:
            if i - x >= 0:
                tmp = t[i] + dp[i - x]
                dp[i] = min(dp[i], tmp)

    return dp[n]
```

f) La complexité temporelle de la fonction `sommeMinDynamique` est en notation grand-O de type $O(n)$, où "n" est la taille de l'entrée, c'est-à-dire le nombre de cases sur le plateau de jeu.

L'algorithme parcourt toutes les cases du plateau de jeu une fois et explore trois options de pas pour chaque case, ce qui donne un nombre total d'opérations proportionnel à la taille du plateau de jeu, soit $O(n)$. La programmation dynamique permet de stocker les résultats intermédiaires pour éviter les calculs répétés, ce qui réduit considérablement la complexité temporelle par rapport à l'approche récursive exponentielle. En conséquence, la complexité temporelle est polynomiale en $O(n)$, ce qui signifie que le temps d'exécution augmente linéairement avec la taille de l'entrée.

Il s'agit d'une amélioration significative par rapport à l'approche exponentielle qui avait une complexité temporelle $O(3^n)$ pour le même problème.

g) Les résultats que vous avez obtenus confirment l'efficacité de l'algorithme optimisé utilisant la programmation dynamique par rapport à l'algorithme non optimisé. Voici une comparaison des deux approches :

Algorithme non optimisé : temps d'exécution de 0.16776394844055176 secondes pour résoudre le problème de taille 31.

Algorithme optimisé : temps d'exécution de 1.811981201171875e-05 secondes pour résoudre le même problème de taille 31.

On peut voir une différence significative dans les temps d'exécution. L'algorithme optimisé est beaucoup plus rapide, avec un temps d'exécution de l'ordre de la dizaine de microsecondes, tandis que l'algorithme non optimisé prend beaucoup plus de temps (environ 0.1677 secondes) pour résoudre le même problème.

Cela confirme que l'approche de programmation dynamique est bien plus efficace pour résoudre ce problème et que le temps d'exécution est en accord avec la complexité temporelle attendue ($O(n)$) de l'algorithme optimisé.